1 Memory Management

Una de las actividades principales del kernel es el manejo de memoria RAM. En lo subsecuente se hará referencia a la memoria RAM solamente como memoria del sistema. Como puede deducirse después de estudiar el arranque de una PC, el sistema operativo debe de proveer una capa de abstracción para el acceso a memoria. La opción mas sencilla es dejar que cada aplicación tenga acceso directo a la memoria y la maneje de manera independiente al resto de las aplicaciones. Sin embargo esto nos enfrenta a diferentes problemáticas. En principio la memoria tendría que manejarse a bajo nivel usando direcciones físicas. El manejo de direcciones físicas dificultaría el desarrollo de aplicaciones. Además abriría la posibilidad de que una aplicación tenga acceso al espacio de memoria de otra aplicación y sobreescribirla o peor aun, sobreescribir el mismo kernel o los servicios del BIOS. Por esta razón se opta en el diseño de kernels por crear una capa de abstracción de la memoria apuntando a incluir las siguientes características:

1. Seguridad
2. Transparencia
3. Eficiencia en el manejo del espacio disponible
4. Manejo de grandes volúmenes de aplicaciones activas
5. Flexibilidad

Estos objetivos intentan dotar al sistema operativo con una capa de abstracción de manejo de memoria que le permita ofrecer un ambiente confiable y eficiente a cada aplicación además de ser lo suficientemente flexible para ejecutar aplicaciones que requieran más memoria de la que físicamente se encuentra disponible. Para poder explicar como se resuelven todos estos puntos, primero se describirá como se encuentra direccionada la memoria física.

1.1 Memory Addressing

La arquitectura actual de procesadores usa un modelos de memoria física conocido como modelo Von Neuman. En este modelo se tiene un solo espacio de memoria que es compartido tanto por el código ejecutable como por los daros. Existen diferentes métodos para manejar la memoria física que son soportados de manera nativa para contribuir con la creación de una capa de abstración para el manejo de memoria por parte del sistema operativo. El controlador de memoria o Memory Management Unit (MMU)de la arquitectura x86 soporta dos modos de operación: Real Mode y Protected Mode. Real Mode es un modo de operación con acceso directo a las direcciones físicas, este modo se incluye para mantener una compatiblidad hacia atrás con los antiguos sistemas operativos y para el arranque usando un bootloader. El modo protegido, que es en el que estaremos trabajando, se describe a continuación. En Protected Mode se utilizan tres tipos de direcciones:

1. Logical Address. Estas direcciones son las manejadas a nivel de lenguajes de bajo nivel como Assembler y se forman de un segment y un offset. Un segment es la direccion inicial del bloque de memoria asignado a un proceso[1] y un offset es el desplazamiento a partir de la dirección inicial contenida en segment. Usualmente solo se debe tener conciencia de estas direcciones cuando se manejan lenguajes o librerías de bajo o medio nivel como C. Estas son las únicas direcciones a las que tenemos acceso en Protected Mode.
2. Linear Address. Este tipo de direcciones etiquetan las zonas de memoria usando un

[1] A partir de ahora se hará referencia a procesos en lugar de aplicaciones, en el siguiente capítulo se aclarará sistemáticamente la diferencia entre ambos.

único índice de 32bits o 64bits dependiendo el tipo de arquitectura. 32 bits pueden diccionar hasta 4GB de memoria mientras que 64 bits pueden direccionar 2B aproximadamente. Esta es la razón por la que sistemas operativos de 32 bytes tienen que usar una modificación especial cuando requieren direccionar memoria mas allá de los 4GB.

3. Physical Address. Esta es la verdadera dirección de una celda de memoria. Esta dirección hace referencia a la señal eléctrica de los pines de la memoria y se representa con un `unsigned int`. De 32 bits o 36 bits para procesadores de 32 bits y 64 bits o para procesadores de 64 bits.

El acceso a memoria a nivel de hardware es manejado por el Memory Arbiter. Esta pieza de hardware permite a cada CPU presente acceder a la memoria así como también a los diferentes dispositivos usando Direct Memory Access sin necesidad de solicitarlo al CPU. Cada CPU requiere un Memory Arbiter dedicado. La physical address es obtenida a nivel de hardware

1.2 Segmentation

Segmentation fue el primer método introducido para incorporar mecanismos de memoria virtual y seguridad en el manejo de memoria. Este mecanismo usa direcciones llamadas Logical Address. En la arquitectura Intel, segmentation es soportado desde el modelo 80286. Para poder entender la manera de calcularlo es necesario revisar la estructura binaria de las Logical Address. El primer paso es obtener la linear address utilizando la logical address. El hardware de la arquitectura x86 calcula de diferente manera la linear address en sus diferentes modos de operación. Para protected mode, el hardware utiliza una estructura de 16 bits para el segment o segment selector y 32 bits para el offset en procesadores de 32 bits. Mientras que el offset es un unsigned int que informa el desplazamiento desde la dirección inicial asignada, el campo de segment tiene una estructura más compleja. La estructura se muestra en la figura .

Los bits están enumerados de derecha a izquierda a modo little indian como es el estandar de la arquitectura x86, siendo el bit menos significativo o LSB por las sigla en inglés de Less Significant Bit el bit de más a la derecha con índice `0` y el bit más significativo o MSB por las siglas en ingles de Most Significant Bit con índice 15 en el caso del segment selector. Los 13 bits mas significativos son llamados index y mantiene el índice de una entrada slmacenada en una Table Descriptor explicada más adelante. El bit 2 indica el tipo de Table Descriptor a la que apunta y por úiltimo los bits 0 y 1 indican el nivel de permiso que tiene el segmento. linux en particular usa solo el valor 0 para kernel mode y 3 para user mode. A cada proceso creado se le otorgan tres segmentos: el code segment donde reside el código ejecutable, el stack segment donde se almacenan valores dinamicamente a manera de pila y el data segment donde se almacenan valores a manera de acceso indisado. La diferencia entre el data segment y el stack segment reside en la manera de acceder a los datos. El procesador mantiene los valores de los distintos segment en los llamados registros de segmento. Son seis registros de segmento, tres de uso particular y tres de uso general. Los tres registros de uso particular son: `ds` para el data segment, `ss` para el stack segment y `ds` para el data segment. Los tres registros de propósito general son: `es`, `fs` y `gs`. Para poder traducir rapidamente de la logical address a la linear address, el sistema operativo utiliza un campo de 8 bytes llamado segment descriptor. Los segment descriptor son almacenados en tablas. Las tablas que almacenan los segment descriptor pueden ser de tipo Global Descriptor Table (GDT) o de tipo Local Descriptor Table (LDT). Usualmente solo se mantiene una GDT por cada procesador presente y su tamaño y dirección son almacenados en el registro gdtr. Las LDT son creadas por los procesos que requieren de reservar mas memoria y su tamaño y dirección son almacenados en el

registro `ldtr`. Cada proceso tiene una sola tabla LDT. Existen varios tipos de segment descriptor pero el hardware mantiene solo tres estructuras físicas. La estructura física para los procesadores de 32 bits se describe en la figura .

La tabla 1 describe los diferentes campos de los segment descriptors.

Como se ha mencionado, existen diferentes tipos de segment descriptor. Aunque físicamente se usa la misma estructura, las banderas indican el tipo de descriptor que se esta almacenando. Este tipo de prácticas solo es válido cuando se trabaja a este nivel de abstracción. En lenguajes de mas alto nivel se deberían crear estructuras de datos con diferentes nombres. La manera de manejar los segment descriptor quedan a discreción del sistema operativo. En eñ caso de linux se manejan cuatro tipos:

- Code Segment Descriptor. Este describe un code segment. La bandera S esta encendida y la bandera Type almacena el valor que representa un code segment descriptor. Las otras banderas dependen del proceso al que pertenece. El segment puede encotrarse en una GDT o una LDT.
- Data Segment Descriptor. Este describe un stack o data segment. La bandera S esta encendida y la bandera Type almacena el valor que representa un data segment descriptor. Para linux, el data y el stack segment no necesitan diferenciarse para calcular su linear address, debido a esto solo se usa un tipo de segment descriptor. El segment puede encontrarse en una GDT o una LDT.
- Task Segment Descriptor. Describe el estado de un proceso. En linux un proceso se suele llamar Task o Thread. Este solo se puede encontrar en la GDT. El campo Type contiene 11 o 9 dependiendo de si el proceso se encuentrs en el CPU o no respectivamente. La bandera S tiene el valor de 0.
- Local Table Table Descriptor. Este descriptor indica que el segmento apunta a una LDT. La bandera S esta apagada y Type tiene el valor de 0. Solo se puede almacenar este tipo en una GDT.

El procesador x86 mantiene registros no programables por cada segment register que almacena los correspondientes segment descriptors. El CPU puede usar estos registros no programables para traducir las logical address en linear address. Estos registros no programables se escriben automáticamente al momento de que un proceso entra en ejecución. Cuando el CPU se ve en la necesidad de calcular la linear address de una logical address que no esta en ejecución utiliza el algoritmo mostrado en la tabla .

Los pasos del algoritmo son los siguientes:

1. El CPU revisa el campo TL del segment selector para determinar si el descriptor esta en la GDT o en la LDT. La dirección de la tabla la obtiene de los registros gdtr o ldtr. Note que el registro ldtr se tiene que reescribir cada que un proceso entra en ejecución. El gdtr se mantiene con el mismo valor desde que el kernel se carga en memoria.
2. El índice del segment descriptor se obtiene multiplicando el valor almacenado en la bandera index del segment selector por 8 que es el tamaño en bytes del segment descriptor y sumándole la dirección de la tabla obtenida en el paso anterior.
3. por último, la linear address se obtiene de sumar el offset de la logical address con la linear address almacenada en el segment descriptor.

En el caso de que la logical address pertenezca a un proceso en ejecución se omiten el primer y segundo paso al tomar el segment descriptor de los registros no programables. La tabla muestra un diagrama de la memoria indicando a donde apuntan las distintas direcciones. Estas operaciones sirven debido a que asumimos que el direccionamiento de memoria es linear y empieza en 0.

Nombre	Descripción
Base	Contiene la linear address del primer byte del segment
G	Granularity Flag. Si esta en 0 el tamaño se expresa en bytes. Si esta en 1 el tamaño se expresa en múltiplos de 4096 bytes
Limit	Contiene el desplazamiento al último byte del segment indicando asi también el tamaño del segmento. Si G tiene el valor de 0 el tamaño va de 1 byte hasta 1 MB. Si G es 1 el tamaño va de 4 KB hasta 4 GB.
S	System Flag. Si es 0, el segment descriptor es un system segment descriptor. Si es 1 es un data o code segment descriptor
Type	El tipo de segment descriptor y sus permisos
DPL	Tipo de privilegio necesario para acceder a este segment descriptor. Si es 0 solo puede accedre el kernel (kernel mode). Si es 3 cualquier proceso puede acceder al segment descriptor (user mode)
P	Present Memory Flag. Indica si: 0, el segmento no está en la memoria o 1 si se encuentra. Este concepto esta relacionado con el swapping entre la memoria y los dispositivos de almacenamiento. El swapping no necesariamente ocurre con segments completos, por lo que sistemas como Linux siempre ponen el valor de 1 ya que nunca hace swapping a todo el segment.
D or B	indica si el offset del segment es de 32 bits cuando esta en 1 o de 16 bits cuando esta en 0
AVL	Bandera de propósito general para el sistema operativo.

Cuadro 1: Flags del Segment Descriptor

En general, los sistemas operativos actuales no hacen uso exteso de este mecanismo debido a que otras arquitecturas diferentes a la x86 no hacen uso de la segmentación. Esta decisión simplifica el trabajo de transportar el sistema operativo a otros dispositivos con arquitecturas diferentes como es el caso de los dispositivos móviles con arquitectura ARM que no soporta segmentación.

1.3 Paging

Paging es un segundo método para crear un modelo virtual de memoria. Este permite que cada segment asignado a un programa no necesite estar en direcciones físicas contiguas como es el caso de usar segmentation. Para matener la eficiencia en el direccionamiento de memoria, el kernel agrupa múltiples linear address en una unidad llamada Page. Todas las linear address dentro de una page apuntan a direcciones físicas contiguas. Esto simplifica el uso de permisos ya que solo se necesita llevar un registro del tipo de permisos por Page en lugar de por linear address. Este mecanismo es llevado a cabo a nivel de hardware por la Paging Unit. La paging unit divide a la memoria RAM en bloques llamados Page frames. El tamaño del page frame es el mismo de una page, lo que permite al CPU colcar una page en cada page frame solamente. La diferencia entre una page y un page frame es que una page es un grupo de linear address y los datos almacenados en cada linear address y un page frame es un espacio de almacenamiento en memoria RAM. Una page puede ser almacenada en otro lugar que no sea la memoria RAM.

La arquitectura x86 utiliza dos métodos para implementar paginación. El primer método, o Paging, divide a la linear address en tres bloques. El primer bloque de los diez bits mas signi-

ficativos, es decir el bit 31 al 22, es llamado Page Directory. Los siguientes diez bits son llamados Page Table y los últimos doce bits son llamados offset. La intención de usar dos niveles es de reducir el uso de memoria por tabla. Este esquema permite mantener una tabla de 2^{10} o 1 MB para almacenar los Table Directory y cada Table Index es cargada a memoria cuando es requerida por un proceso. Tanto la Table Directory como la Table Index tienen la misma estructura. Este método permite Pages de 4KB de tamaño[2]. Esta estructura es mostrada en la tabla 2.

Las funciones de las banderas se explican a continuación. Los nuevos conceptos que aparacen en la tabla se iran explicando a lo largo del resto del capítulo.

El concepto de swapping mencionado en las banderas Present, Dirty y Accessed es parte del me anismo de memoria virtual por el que fue concebido el Paging. Sin olvidar la seguridad. Swapping es el mecanismo que intercambia una page contenida dentro de un page frame a un dispositivo de almacenmiento secundario y pone ptra page en su lugar. Esto se utiliza para extender el espacio disponible en memoria en ciertos casos. El sistema de memoria virtual se concibio debido a que, desde la creación de las computadoras, la memoria RAM siempre ha sido mas cara que los dispositivos de almacenamiento magnético. El sistema de memoria virtual nos permite extender la memoria disponible para procesos que requieran una cantidad inferior a la memoria física existente. Este método no sirve si se requiere de un bloque contiguo que exceda el tamaño de la memoria física. La desventaja de este método es el que el acceso a dispositivos de almacenamiento es varias veces mas lento que a memoria. Esto provoca que un sistema que haga mucho swapping vea su rendimiento reducido ya que involucra al menos tres acciones: solicitud de una page, excepción de page fault por la paging unit, swapping por el sistema operativo. El sistema operativo se encarga de decidir que page intercambia con una en el dispositivo de almacenamiento al ver las banderas Dirty y Accessed. El sistema operativo usa la page es la menos usada como candidata a swapping.

El segundo método de paging se conoce como extended paging. Este método se incorporó en la arquitectura x86 a partir del modelo Pentium. En extended paging se usan pages de 4 MB en lugar de 4 KB. Esto se logra dividiendo la logical address en dos niveles en lugar de tres. Los primeros 10 bits mas significativos son el Page Directory y los siguientes 22 son el offset de la physical address. Los bits de los campos del Page Directory siguen siendo los mismos excepto que el campo de la physical address solo se toman los primeros 10 bits mas significativos debido a que estamos manejando múltiplos de 4 MB y los 22 bits menos significativos de estos múltiplos siempre se encuentran apagados. Cual método usar depende de la aplicación. El método de Paging decrece en rendimiento cuando un proceso solicita bloques de memoria grandes. Esto es si el proceso accede constantemente a las diferentes pages que componen el bloque de memoria que solicitó, lo que puede provocar múltiples excepciones de page fault. Extended Paging presenta un mejor rendimiento en estos casos al reducir el número de excepciones para este tipo de procesos. En otros casos Paging presenta un mejor rendimiento promedio. Extended paging tambien presenta una ventaja en el uso de cache explicado mas adelante.

Un problema que se presentó previo a la aparición de procesadores de 64 bits es cuando se requeria direccionar mas de 4 GB de memoria física ya que solo se manejaba una dirección de 32 bits[3]. Este problema aparece primero en servidores y mas adelante en equipos domésticos con el abaratamiento de la memoria RAM. Para solucionar esto, Intel introduce el mecanismo llamado Physical Address Extension (PAE) Pa-

[2] el offset usa 12 bits, $2^12 = 4096$

[3] $2^{32} \simeq 4$ GB

Nombre	Descripción
Present	Tiene el valors 1 si la Page se encuentra en memoria. Si es 0 el resto de los campos no tienes significado. En caso que un proceso solicite esta page la paging unit levanta una excepción llamada page fault.
Physical Address	Los siguientes 20 bits representan las physical address en una Page Table. Solo se necesitan 20 bits en lugar de 32 porque las pages son de 4 KB de tamaño. Por lo tanto la dirección inicial de una page es un múltiplo de 4096 y los 12 bits menos significativos estarán apagados. En una Page Directory apunta a la Page Table que contiene la physical address.
Accessed	Toma el valor de 1 cuando la paging unit la usa para calcular una dirección solicitada. Este campo es usando junto con Present para implementar swapping.
Dirty	Toma el valor de 1 cuando alguna localidad de una Page Table es escrita. También se usa para swapping
Read/Write	Permisos de la page table, puede ser Read o Read/Write
User/Supervisor	Indíca el nivel de permiso requerido para pode acceder a la page.
PCD y PWT	Estas dos banderas indican como debe ser administrada la page cuando esta en memoria cache.
Page size	Indica el método de paginación
TLB	Esta bandera también se usa para el manejo de cache.

Cuadro 2: Flags del Table Directory

ging a partir de los procesadores Pentium Pro. Para esto, aumenta el número de pines de la memoria de 32 a 36 e introduce un nuevo nivel de paging creando una tabla especial con entradas de 64 bits que puedan almacenar los 36 bits necesarios. Note que se usan 64 bits en lugar de los 36 ya que los procesadores solo manejan localidades con tamaño múltiplo del tamaño el bus, es decir 32 bits en este caso.

Para procesadores de 64 bits se mantuvo el tamaño de las pages en 4KB excepto en ciertas arquitecturas como la alpha de o la ia64 de Intel para servidores donde el tamaño es variable. Para mantener la eficiencia en estas arquitecturas se decidió introducir mas niveles de paging. Para la versión de 64 bits de la arquitectura x86 (x86_64) se manejan 4 niveles de paging, separando la linear address en cuatro bloques de pages de 9 bits cada uno y un offset de 12 bits (4 KB). La cantidad de niveles varia de arquitectura en arquitectura, un ejemplo de esto es la arquitectura PowerPC de IBM usada en las antiguas computadoras apple y la Playstation 3 que usan tres niveles de paginación separando la linear address en dos bloques de pages de 10 bits, un bloque de page de 9 bits y un offset de 12 bits. Note que ninguna de las arquitecturas mencionadas usan 64 bits para la linear address. Por el momento, se consideró que 46 bits en la arquitectura x86_64 pueden satisfacer la demanda en direccionamiento de memoria. Esto equivale a direccionar 2^{46} direcciones físicas lo que es al rededor de.

1.4 Caching

La velocidad de la memoria es hasta ahora menor que la de los procesadores. Esto implica que el procesador queda esperando a la memoria por la transferencia de una localidad. La di-

ferencia de velocidades crea un cuello de botella. Para poder reducir este efecto, se creó la memoria cache. Este tipo de memoria es mas rápida debido a que es de tipo Static RAM en lugar de la usada comunmente denominada Dynamic RAM. La cache se coloca dentro del dado el procesador dandole mayor ventaja al estar mas cerca del CPU. La implementación de este tipo de memoria usa el llamado Principio de Localidad. Este principio hace referencia a que despues de acceder a una localidad de memoria, es altamente probable que se enseguida se acceda a alguna localidad cercana. Por esto, la cache esta formada por varias unidades llamadas lines. Cada line puede almacenar varias localidades de memoria. La transferencia entre la memoria principal y la cache sucede en rafagas donde se transmiten varias localidades juntas. El modo de operación es similar al de Paging. La cache es admininstrada por el cache controller. El cache controller almacena un conjunto de bits llamado Tag para identificar las localidades de memoria de cada line. Cuando el procesador requiere una localidad, el cache controller revisa el tag asociado y lo compara con los tag de cada line. Si lo encuentra se tiene un cache hit, en caso contrario se tiene un cache miss. Cuando ocurre un cache hit existen dos comportamientos dependiendo de la operación a realizar. Cuando es de lectura, el cache controller transfiere la localidad al CPU sin acceder a la memoria RAM. En el caso de que la operación sea de escritura existen dos estrategias: write through y write back. En write through, el cache controller escribe la line tanto en la cache como en la memoria principal. En write back, el cache controller solo escribe la line en la cache, esperando hasta que exista una solicitud de flush por parte del CPU para escribir las lines modificadas a RAM. La solicitud flush ocurre principalmente en un evento de cache miss.

Cuando ocurre un cache miss, se realiza un flush de ser necesario y se sobrescribe una linea por la solicitada por el CPU. En equipos con multiples nucleos con cache propia y memoria RAM compartida, el cache controller siempre tiene que revisar en una operación de escritura las cache de los otros procesadores para actualizarla en caso de ser necesario. A esto se le llama cache snooping.

La memoria cache fue implementada desde los procesadores Pentium de Intel. A este tipo de cache se le conoce como L1-cache. Modelos recientes implementan varios niveles de cache (L2, L3, etc) donde cada nivel es mas lento que el anterior pero mas rápido que la memoria RAM. El manejo de los distintos niveles es dejado a nivel de hardware por lo que sistemas operativos como Linux asumen solo un nivel de cache.

Existe una segunda cache usada para acelerar el cálculo de las pysichal address llamado Translation Lookaside Buffer (TLB). Cuando una pysichal address es calculada, se utilizan la Page Tables la primera vez y el resultado se guarda en la TLB para que en posteriores accesos se utilize la dirección almacenada en la TLB en lugar de hacer todo el cálculo.